Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties

Narges Khakpour[1], Oliver Schwarz[2,1], and Mads Dam[1]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] SICS Swedish ICT, Kista, Sweden
`{nargeskh,oschwarz,mfd}@kth.se`

Abstract. In this paper, we formally verify security properties of the ARMv7 Instruction Set Architecture (ISA) for user mode executions. To obtain guarantees that arbitrary (and unknown) user processes are able to run isolated from privileged software and other user processes, instruction level noninterference and integrity properties are provided, along with proofs that transitions to privileged modes can only occur in a controlled manner. This work establishes a main requirement for operating system and hypervisor verification, as demonstrated for the PROSPER separation kernel. The proof is performed in the HOL4 theorem prover, taking the Cambridge model of ARM as basis. To this end, a proof tool has been developed, which assists the verification of relational state predicates semi-automatically.

Keywords: ARM instruction set, noninterference, user mode execution, kernel security, theorem proving

1 Introduction

The ability to execute application software in a manner which is isolated from other application software running on a shared processing platform is an essential prerequisite for security. This allows user applications or virtual machines to coexist without violating confidentiality or integrity of critical data, it allows critical system resources to be protected from user manipulation, it can help to prevent fault propagation, and it can be used to save costly hardware that might otherwise be needed to provide physical separation.

Isolation is typically provided by a mix of hardware and software. A memory management unit (MMU) may be used to provide basic memory protection, and the processor may be equipped with multiple privilege levels, running application programs as userland processes and kernel routines at privileged levels, with additional abilities to access and configure critical parts of the processor, the MMU, and various storage/display/peripheral devices attached to the processor.

In such a setting, isolation is a result of the correct interplay between hardware and kernel. It is the responsibility of the kernel to correctly manipulate the processor state to achieve the desired effects, whatever they may be (context switching, logging, fault management, device management, etc). It is the responsibility of the processing hardware to correctly implement the partitioning safeguards and mode transition conventions assumed by the kernel. For security, the kernel and the processor must both be correct and agree on their mode of interaction. Most formal kernel analyses in the literature [7,12,13,15,18] address the kernel software itself, in source or binary form, and leave the properties of the instruction set architecture (ISA) to be handled by fiat. Our contribution is to suggest a possible approach, including tool support, for performing the ISA specific security analysis, specifically for user mode execution.

We have identified two main concerns.

First, an implicit contract must exist which stipulates the "region of influence/dependency" of userland processes. That is, in a given user mode processor/MMU configuration it must be determined which memory locations and (control) registers can be read or written, or, in a more fine grained analysis, how information is able to flow to or from specific parts of the processor and the memory. User processes must be constrained in accessing or otherwise being influenced by critical resources of the kernel or of other user processes. This is not trivial. For instance, as shown by Duflot et al. [9], on some x86 processors it is possible for low-privilege code to overwrite higher privilege code by writing to an address that usually refers to the video card. To enable this attack, it suffices to first flip a configuration bit usually accessible from the low privilege level.

Second, kernel code relies on a set of mode switching conventions, for instance on ARM that program status registers and relevant user registers (including the program counter) are properly banked, the program counter is updated to point at the correct location in the vector table, and so on. If these conventions are not established by the processor and adhered to by the kernel, it may be possible for userland processes to induce various sorts of malicious behavior, for instance by letting a handler's link register point to a foreign address.

Performing this analysis is not trivial, particularly not if information flow is to be taken into account, as is done in this paper. All instructions, error conditions, and user to privileged mode transitions must be considered. The number of instructions is high and in modern processors a single instruction can involve a large number (order of 20-30) of atomic register or memory accesses.

In this paper, we identify and prove several partitioning-related properties of the ARMv7 ISA specification [2,3] addressing user mode execution and mode switching. The first is an instruction level noninterference property related to the non-infiltration property in [12] stating that the behavior of an ARMv7 processor in user mode only depends on its accessible resources, mostly user registers, MMU configurations and the memory allocated to that process. The second, corresponding to the non-exfiltration property of [12], is an integrity property stating that, again while in user mode, the processor is unable to modify protected resources. A third set of properties concerns mode switching conventions. These properties have been applied in the PROSPER project [5] to verify isolation for the PROSPER separation kernel [8]. The PROSPER project aims at producing and verifying a fully functional secure hypervisor for embedded systems, providing services such as guest isolation, so that only explicitly allowed communication occurs.

Our proof uses the HOL4 [4] model of ARM, developed at Cambridge by Fox et al. [10]. We extend this model by simple memory protection. The ARMv7 ISA properties outlined above are formalized and proved. To make the quite sizable proof task feasible, we have developed a helper tool based on relational Hoare logic, that is able to automate significant parts of the proof.

To the best of our knowledge our work represents the first formalized analysis of the ARMv7 ISA. Others, specifically the Cambridge HOL4 group, have developed various helper tools for assembling, disassembling, executing, and managing ARM machine code and the HOL4 ARM ISA model [10,16]. Also, the HOL4 ARM model has been used in several verification exercises in the literature, on software fault isolation (SFI) [22] and on the extension of the seL4 verification work [13] from C to binary level [20]. However, we have not yet seen general correctness properties formalized and verified for ARM at the ISA level. In fact, we believe the type of analysis presented here can be useful beyond kernel verification. For instance, formalized security properties can be useful to both improve the usefulness and precision of ISA specifications, and to enable developers obtain a concise description of secure configurations, without manual consideration of extensive architecture specifications.

2 The Formal Specification of ARM

We use Fox et al's monadic HOL4 model [10] of the ARMv7 ISA. This model covers the ARM, Thumb and ThumbEE instruction sets, comprising 81 instructions for branching, memory access, data processing, co-processor access, status access, and miscellaneous functionality. Figure 1 shows a simplified definition of an ARM state in this model. The function `psrs` returns the value of a processor state register (of type `ARMpsr`). The processor state registers include the current program status register, `CPSR`, in addition to the banked psrs `SPSR_m` for each privileged mode `m`, except for system mode. Program status registers encode arithmetic flags, the processor mode `M`, interrupt masks (`I` for ordinary and `F` for fast interrupts) and instruction encoding. The ARMv7 core provides seven processor modes: one non-privileged user mode `usr`, and six privileged modes (`abt`,`fiq`,`irq`,`svc`,`und`,`sys`), activated when an exception (such as an interrupt) is invoked. Variants with the TrustZone extension [1] also have a monitor

```
arm-state = <| psrs         : PSRName -> ARMpsr;
               regs         : RName -> word32;
               memory       : word32 -> word8;
               coproc       : coprocessors;
               accesses     : memory_access list;
               misc         : Monitors # ARMinfo # bool # bool |>;
```

Fig. 1. The ARM state in HOL4

mode. However, this has to be invoked from a privileged mode and we consider its usage out of scope of this paper.

The function `regs` takes a register name and returns its value. The ARM registers include sixteen general purpose registers (`r0`-`r15`) that are available from all modes in addition to the banked registers of each privileged mode (except of `sys`) that are available only in that mode. Among the user registers, register `r13` functions as stack pointer `SP`, register `r14` as link register `LR` and register `r15` as program counter `PC`.

The function `memory` reads a byte (`word8`) from an address (`word32`). The field `coproc` represents those coprocessor registers in `CP14` and `CP15` that implicitly influence execution. The coprocessor registers central for this work are registers `SCTLR` , `TTBR0` and `DACR` of coprocessor 15. They, together with the page table, are used to configure the MMU. The field `misc` represents the exclusive monitors used for synchronization purposes, general information about the state, e.g. the architecture version, if the system is waiting for an interrupt etc, and `accesses` records the accesses to the memory.

A *computation* in the monadic HOL4 ARM model is a term of the following (slightly beautified) type

$$\alpha\ \mathtt{M} = \mathtt{arm\_state} \mapsto (\alpha, \mathtt{arm\_state})\ \mathtt{error\_option}.$$

where `error_option` is a datatype defined as follows:

```
(α,β) error_option = ValueState of α => β
                   | Error of string
```

Computations act on a state `arm_state` and return either `ValueState` a s, a new state s of type `arm_state` along with a return value a of type α, or an error e. The unpredictable computations, i.e., those that are underspecified by the ARM specification return an error. The monad unit `constT` injects a value into a computation, i.e. `constT` a s = `ValueState` a s, while binding is a sequential composition operation

$$\begin{aligned} f_1 \gg\!=_e f_2 = \lambda s.\mathtt{case}\ f_1 s\ \mathtt{of}\ \mathtt{Error}\ c \rightarrow \mathtt{Error}\ c \\ \|\ \mathtt{ValueState}\ a\ s' \rightarrow \\ \mathtt{if}\ e\ s'\ \mathtt{then}\ f_2\ a\ s'\ \mathtt{else}\ f_1\ s. \end{aligned}$$

That is, if e holds in the final state of f_1, the return value of f_1 is passed to f_2 as the input parameter, otherwise f_2 is not executed.

$$\begin{aligned}
&\texttt{errorT}\ a = \texttt{Error}\ \ a\\
&\texttt{condT}\ e\ f = \texttt{if}\ e\ \texttt{then}\ f\ \texttt{else}\ \texttt{constT}\ ()\\
&\texttt{if}\ e\ \texttt{then}\ f_1\ \texttt{else} f_2 = \lambda s.\texttt{if}\ e\ s\ \texttt{then}\ f_1\ s\ \texttt{else}\ f_2\ s\\
&f_1\ |||_e\ f_2 = f_1 \gg\!=_e (\lambda x. f_2 \gg\!=_e (\lambda y.\texttt{constT}\ (x, y)))\\
&\texttt{forT}_e\ l\ h\ f = \texttt{if}\ l > h\ \texttt{then}\ \texttt{constT}\ []\\
&\qquad\qquad\qquad\texttt{else}\ ((f\ l) \gg\!=_e (\lambda r.\texttt{forT}_e\ (l+1)\ h\ f \gg\!=_e (\lambda l.\texttt{constT}\ r :: l)))
\end{aligned}$$

Fig. 2. Auxiliary monad operations

In addition to unit and binding, the ARM monadic specification uses standard constructs for lambda, let, and cases, as well as the monad operations parallel composition ($f_1\ |||_e\ f_2$), positive conditional ($\texttt{condT}\ e\ f$), full conditional ($\texttt{if}\ e\ \texttt{then}\ f_1\ \texttt{else}\ f_2$), error ($\texttt{errorT}\ a$), and an iterator ($\texttt{forT}_e\ l\ h\ f$), (inductively) defined in Figure 2.

3 Memory Management

The Memory Management Unit (MMU) enforces memory access policies and is therefore important for isolation. MMU configurations consist of page tables in memory and dedicated registers of `CP15`. Specific to ARM is the possibility of partitioning pages into collections of memory regions, so-called *domains*. The theorems in this paper are based on the concrete MMU configurations (memory ranges, the page table setup etc.) used in the PROSPER kernel. The coprocessor registers involved are `SCTLR`, `TTBR0` and `DACR`. The `SCTLR` register determines whether the MMU is enabled, `TTBR0` contains the base address of the page table, and `DACR` manages the ARM domains.

MMU Extension The evaluation function `permitted` takes as parameters a byte address, a flag indicating whether reading or writing access is to be evaluated, the values of `SCTLR`, `TTBR0` and `DACR`, a flag indicating whether permissions are to be checked against a privileged mode, and the memory containing the page tables. The pair of booleans returned by `permitted` states whether the access permission on the specified byte is defined in the given configuration and the outcome of that decision (`true` if access is granted). The PROSPER kernel uses a basic version of `permitted`, supporting one-level page tables without address translation, but including the interpretation of ARM domains. It is shown that `permitted` is defined for all addresses in all reachable states.

The history of memory accesses is tracked in the `accesses` field of the machine state, allowing to compute the set of memory pages accessed by an instruction. To stop computation after the first access violation, $\gg\!=_{\texttt{nav}}$ has been chosen as standard binding operator, where `nav s` ("no access violation") is `true` if and only if there is no entry in the access list of machine state `s` that causes `permitted` to return a negative answer int the current configuration of `s`. The recording of an access always happens before the access itself.

```
next irpt s =
(clear_alist ≫=_nav
 (λu. if irpt = NoInterrupt then
         waiting_for_interrupt ≫=_nav
         (λwfi. condT (¬wfi)
                   (fetch_instruction ≫=_T
                   (λ(opc, ins). is_viol ≫=_T (λav. clear_alist ≫=_nav
                   (λu. if av then prefetch_abort
                        else
                           (execute ins ≫=_T (λu. is_viol ≫=_T
                           (λav. condT av
                                    (clear_alist ≫=_nav
                                    (λu. data_abort)))))))))
      else take_exception irpt ≫=_nav (λu. clear_wait_for_irpt))) s
```

Fig. 3. The `next` computation.

The instruction execution function `next` (see Figure 3) takes an exception/interrupt flag `irpt` and a state `s` and produces the consequent state, by either initiating the demanded exception or by fetching and executing the next instruction pointed to by the `PC` in `s`. If an access violation is recorded after instruction fetching or execution, a prefetch or data abort exception (respectively) is initiated. The access list is cleared between the single steps, preventing the execution from halting and instead proceeding with exception handling. Occasionally, the unconditional binding $\gg=_T$ is used.

MMU Configuration Let `accessible i a` express that address `a` is readable and writable by user process `i`. The predicate `mmu_setup i s` holds if and only if (i) state `s` implements the desired access policy for process `i`, (ii) no MMU configuration for any address is underspecified, and (iii) none of the active page tables in `s` (represented by the address set `page_table_adds s`) is accessible according to the policy.

```
mmu_setup i s = ∀add, is_write, u, p.
   (u,p) = permitted add is_write (mmu_registers s) F s.memory
 ⇒ u ∧ ((accessible a i) ⇔ p)
      ∧ (a ∈ (page_table_adds s) ⇒ ¬(accessible a i))
```

4 Security Properties

We next turn to formalizing the instruction level partitioning properties. For user mode execution we formulate the requirements in terms of non-infiltration and non-exfiltration properties (cf. [12]), adapted to our setting.

Our model does not include caches, timing or hardware extensions such as TrustZone or virtualization support. Devices are not part of the model either; however, interrupts and other exceptions are taken into account, apart from fast interrupts and resets. Accordingly, the `fiq` and `mon` modes are outside of our analysis. As discussed, the chosen memory configuration is specific to the PROSPER project. Consequences of a limited coprocessor model and underspecified instructions are discussed in Section 8.

4.1 Non-infiltration

Confidentiality of the kernel and neighboring user processes is guaranteed by non-infiltration, a noninterference-like property at the user mode single instruction level. Consider two machine states in user mode that are *low equivalent* in the sense that the two states agree on the resources (registers and memory locations) that are permitted to influence user mode execution, but do not necessarily agree on other resources. Non-infiltration holds if the poststates, after execution of one instruction, remain low equivalent (or produce the same error).

Theorem 1. *Non-infiltration*

```
∀s1, s2, i, irpt.  mode s1 = mode s2 = usr ∧ bisim i s1 s2
⇒ (∃t1, t2. next irpt s1 = ValueState () t1
          ∧  next irpt s2 = ValueState () t2 ∧ bisim i t1 t2)
∨  (∃e. next irpt s1 = Error e ∧ next irpt s2 = Error e)
```

The relation `bisim` is the low equivalence relation. User mode processes are allowed to be influenced by the user mode registers, the memory assigned to them, the CPSR, the coprocessors, pending access violations and the `misc` state component. Exclusive monitors (as field of `misc`) can inherently influence and be influenced by user mode software and need thus to be cleared by kernels on context switches.

```
bisim i s1 s2 =
    mmu_setup i s1 ∧ mmu_setup i s2 ∧ (equal_user_regs s1 s2)
  ∧ (∀a. (accessible i a) ⇒ (s1.memory a = s2.memory a))
  ∧ (s1.psrs(CPSR)= s2.psrs(CPSR)) ∧ (s1.coproc.state = s2.coproc.state)
  ∧ (nav s1 = nav s2) ∧ (s1.misc = s2.misc)
  ∧ s1.psrs(spsr_(mode s1)) = s2.psrs(spsr_(mode s2))
  ∧ s1.regs(lr_(mode s1)) = s2.regs(lr_(mode s2))
```

The two last items have been included to assure that SPSR and link register (of a possibly privileged poststate) only depend on resources allowed to influence user mode execution as well, so that they can actually be restored later on.

4.2 Non-exfiltration

Non-exfiltration guarantees the integrity of resources foreign to the active user process. It expresses that, given an MMU setup for user process `i` active, the execution of a single instruction in user mode will not modify any other resources but those considered to be modifiable by `i`.

Theorem 2. *Non-exfiltration*

```
∀s, t, i, irpt. mode s = usr ∧ mmu_setup i s
  ∧ next irpt s = ValueState () t ⇒ unmodified i s t
```

Here, `unmodified` expresses the desired relation between the prestate `s` and the poststate `t` of an active process `i`. We require that coprocessors, the fast interrupt flag and any memory not belonging to `i` remain unchanged. The only registers allowed to change are the `CPSR`, the user mode registers, and the `PSR` and the link register of the mode in `t`. The interrupt flag of the `CPSR` is not modified when staying in user mode.

```
unmodified i s t =
    (s.coproc = t.coproc) ∧ (s.psrs(CPSR).F = t.psrs(CPSR).F)
  ∧ (∀a. ¬(accessible i a) ⇒ (s.memory a = t.memory a))
  ∧ ((mode s ∈ {usr, mode t} ∧ mode t ∈ {usr, fiq, irq, svc, abt, und})
   ⇒( (∀reg. reg ∉ accessible_regs(mode t) ⇒ s.regs(reg) = t.regs(reg))
     ∧ (∀psr. psr ∉ {CPSR, spsr_(mode t)} ⇒ s.psrs(psr) = t.psrs(psr))
     ∧ (mode t = usr ⇒((s.psrs(CPSR)).I = (t.psrs(CPSR)).I))))
```

4.3 Switching to Privileged Modes

Secure user mode execution is not by itself sufficient. It is also necessary to consider transitions to privileged modes to prevent user processes from privileged execution rights. No user process should be able to effect a mode change with the `PC` set to a memory location of his choice. Instead, all entry points into privileged modes should be in the exception vector table. Similarly, even though user processes are allowed to choose a different endianness for their own execution, that should not influence the interpretation of the system handlers when switching back to privileged mode. Theorem 3 covers those additional constraints.

Theorem 3. *Privileged Constraints*

```
∀s, t, i, irpt. mode s = usr ∧ mmu_setup i s
  ∧ next irpt s = ValueState () t ⇒ priv_const s t
```

Besides the above properties, the relation `priv_const` lists the reachable processor modes[3] and assures that interrupts are masked when entering a privileged mode. Also, status register flags regarded as unwritable will be copied from the `CPSR` in prestate `s` to the `SPSR` in poststate `t`. This guarantees that a kernel can restore the saved program status register without further modifications when jumping back to the user process. Otherwise, user processes would be able to make the kernel enable/disable interrupts or change their execution mode. All access violations, if there were any, will have been handled (`nav t`).

[3] Monitor and system mode can only be reached from another privileged mode.

```
priv_const s t =
  mode t ∈  {usr, fiq, irq, svc, abt, und}
  ∧ (mode t ≠ usr ⇒
     (   t.regs(PC) ∈ vt_adds(vt_base s, mode t) ∧ nav t
      ∧ (t.psrs(CPSR)).(I, J, IT, E) = (T, F, 0w, endianess s)
      ∧ (t.psrs(spsr_(mode t))).(M, I, F)
           = (usr, (s.psrs(CPSR)).I, (s.psrs(CPSR)).F)))
```

4.4 Link Register Contents in Supervisor Mode

Upon reception of a software interrupt, exception handlers in the invoked supervisor mode (svc) often need to analyze the calling instruction, in order to determine the software interrupt number for example. Therefore, verification might require assertions that the memory location pointed to by the link register actually does belong to the user process which caused the switch to supervisor mode. Formally, when going from state s in user mode to state t in supervisor mode, it is required that the svc-link register of t (i) is equal to the PC of s plus an instruction set dependent offset and (ii) corrected by the offset, points to an aligned word that is readable in t (independent of the mode). Note that offset and width of the word depend on the instruction set used by the user process, not on the one used by the handler.

Theorem 4. *Link Register Constraints*

```
∀s, t, i, irpt, lr. mode s = usr ∧ mmu_setup i s
  ∧ next irpt s = ValueState () t ∧ mode t = svc ∧ lr = t.regs(LR_svc)
⇒ lr = s.regs(PC) + offset s
  ∧ ((t.psrs(SPSR_svc)).T ⇒ aligned_word_readable t T (lr - 2w))
  ∧ (¬(t.psrs(SPSR_svc)).T ∧ ¬(t.psrs(SPSR_svc)).J
         ⇒ aligned_word_readable t F (lr - 4w))
```

Here, aligned_word_readable s b add states that the aligned word referred to by add is readable in s. Dependent on whether b is true or false, word width and alignment are 16 or 32 bit.

4.5 Safe User Mode Execution

The final aim is to guarantee that as long as the machine is executing in user mode, it causes no noninterference or integrity violations. Let $\mathtt{s_1} \rightsquigarrow \mathtt{s_n}$ denote a sequence of next computations $\mathtt{s_1} \rightarrow \mathtt{s_2} \rightarrow .... \rightarrow \mathtt{s_n}$ in user mode, i.e. mode $\mathtt{s_i} = \mathtt{usr}$, $1 \leq i < n$ and mode $\mathtt{s_n} \neq \mathtt{usr}$. The following theorem assures the safe execution and safe mode switching of a user process.

Theorem 5. *Let* $\mathtt{s_1} \rightsquigarrow \mathtt{s_n}$ *and* mmu_setup i $\mathtt{s_1}$*, (i) if* $\mathtt{s'_1} \rightsquigarrow \mathtt{s'_n}$ *and* bisim i $\mathtt{s_1}$ $\mathtt{s'_1}$ *then* bisim i $\mathtt{s_n}$ $\mathtt{s'_n}$*, (ii)* unmodified i $\mathtt{s_1}$ $\mathtt{s_n}$*, and (iii)* priv_const $\mathtt{s_{n-1}}$ $\mathtt{s_n}$.

The proof of (i) and (ii) is an easy induction on n using theorems 1 and 2. Item (iii) follows from Theorem 3.

$$\text{errorTR}\frac{}{\{\texttt{errorT}\ a : \texttt{R\_m} \to \texttt{R\_m}\}} \qquad \text{constTR}\frac{}{\{\texttt{constT}\ a : \texttt{R\_m} \to \texttt{R\_m}\}}$$

$$\text{condTR}\frac{\{f : \texttt{R\_m} \to \texttt{R\_m}\}}{\{\texttt{condT}\ \psi\ f : \texttt{R\_m} \to \texttt{R\_m}\}} \qquad \text{forTR}\frac{\{f : \texttt{R\_m} \to \texttt{R\_m}\}}{\{\texttt{forT}_{\texttt{nav}}\ l\ h\ f : \texttt{R\_m} \to \texttt{R\_m}\}}$$

$$\text{conR}\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\} \quad \{f' : \texttt{R\_m} \to \texttt{R\_n}\}}{\{\texttt{if}\ \psi\ \texttt{then}\ f\ \texttt{else}\ f' : \texttt{R\_m} \to \texttt{R\_n}\}}$$

$$\text{widenR}\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\}}{\{f : \texttt{R\_m} \to \texttt{R\_(n,k)}\}} \qquad \text{absR}\frac{\forall y.\{f\ y : \texttt{R\_m} \to \texttt{R\_n}\}}{\{\lambda y.f : \texttt{R\_m} \to \texttt{R\_n}\}}$$

$$\text{seqTR}\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\} \quad \{f' : \texttt{R\_n} \to \texttt{R\_k}\} \quad (\texttt{m} = \texttt{n}) \vee (\texttt{n} = \texttt{k})}{\{f \gg\!=_{\texttt{nav}} f' : \texttt{R\_m} \to \texttt{R\_(n,k)}\}}$$

$$\text{parTR}\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\} \quad \{f' : \texttt{R\_n} \to \texttt{R\_k}\} \quad (\texttt{m} = \texttt{n}) \vee (\texttt{n} = \texttt{k})}{\{f|||_{\texttt{nav}} f' : \texttt{R\_m} \to \texttt{R\_(n,k)}\}}$$

Fig. 4. Relational inference rules

5 The Logic Framework

Considering the size and complexity of the ARM model and the instruction set, to prove the properties of the previous section tool support is essential. In this section we present proof rules for relational and invariant reasoning that help to automate the proof.

Non-infiltration The proof uses a relational Hoare logic based on assertions $\{f\texttt{:R} \to \texttt{R'}\}$ defined as follows:

$$\begin{aligned}\{f\texttt{:R} \to \texttt{R'}\} = \forall s_1,s_2.\ \texttt{R}\ s_1\ s_2 \Rightarrow\ & (\exists a,t_1,t_2.\ f\ s_1 = \texttt{ValueState}\ a\ t_1 \wedge \\ & \quad f\ s_2 = \texttt{ValueState}\ a\ t_2 \wedge \texttt{R'}\ t_1\ t_2) \\ & \vee(\exists e. f\ s_1 = \texttt{Error}\ e \wedge f\ s_2 = \texttt{Error}\ e)\end{aligned}$$

The judgment asserts that, if started in prestates s_1, s_2 related by prerelation R, either the executions of the monadic computation f return identical values a with poststates t_1, t_2 related by postrelation R', or else they both return the same error e.

For the analysis it suffices to consider a fixed set of relations

$$\texttt{R\_m} = \lambda s_1.\lambda s_2.\texttt{bisim i}\ s_1\ s_2 \wedge \texttt{mode}\ s_1 = \texttt{m} \wedge \texttt{mode}\ s_2 = \texttt{m}$$

or $\texttt{R\_(n,m)} = \texttt{R\_n} \cup \texttt{R\_m}$.

Figure 4 shows the relational logic inference rules. The inference system is incomplete, but sufficient for our purpose. A relation R_m is preserved by errorT and constT (rules constTR and errorTR), and if a computation preserves one of the R_m relations then that computation can be used in a conditional or a *for* loop as well (condTR, conR and forTR). The rule widenR and absR are used to weaken the postrelation and reason about lambda computations, respectively. The rule seqTR states that the postrelation of $f \gg\!=_{\texttt{nav}} f'$ is the union of the

$$\text{errorTI}\,\frac{}{\mathtt{INV}\langle \mathtt{errorT}\ a, \mathtt{Q}, \mathtt{P}\rangle} \qquad \text{constTI}\,\frac{\mathtt{refl\ P}}{\mathtt{INV}\langle \mathtt{constT}\ c, \mathtt{Q}, \mathtt{P}\rangle}$$

$$\text{condTI}\,\frac{\mathtt{refl\ P}\quad \mathtt{INV}\langle f, \mathtt{Q}, \mathtt{P}\rangle}{\mathtt{INV}\langle \mathtt{condT}\ e\ f, \mathtt{Q}, \mathtt{P}\rangle} \qquad \text{forTI}\,\frac{\mathtt{refl\ P}\quad \mathtt{trans\ P}\quad \mathtt{INV}\langle f, \mathtt{Q}, \mathtt{P}\rangle}{\mathtt{INV}\langle \mathtt{forT}_e\ l\ h\ f, \mathtt{Q}, \mathtt{P}\rangle}$$

$$\text{conRI}\,\frac{\mathtt{INV}\langle f, \mathtt{Q}, \mathtt{P}\rangle \quad \mathtt{INV}\langle f', \mathtt{Q}, \mathtt{P}\rangle}{\mathtt{INV}\langle \mathtt{if}\ \psi\ \mathtt{then}\ f\ \mathtt{else}\ f', \mathtt{Q}, \mathtt{P}\rangle}$$

$$\text{absI}\,\frac{\forall y.\mathtt{INV}\langle f\ y, \mathtt{Q}, \mathtt{P}\rangle}{\mathtt{INV}\langle \lambda y.f, \mathtt{Q}, \mathtt{P}\rangle} \qquad \text{seqTI}\,\frac{\mathtt{INV}\langle f, \mathtt{Q}, \mathtt{P}\rangle \quad \mathtt{INV}\langle f', \mathtt{Q}, \mathtt{P}\rangle \quad \mathtt{trans\ P}}{\mathtt{INV}\langle f \gg=_e f', \mathtt{Q}, \mathtt{P}\rangle}$$

$$\text{parTI}\,\frac{\mathtt{INV}\langle f, \mathtt{Q}, \mathtt{P}\rangle \quad \mathtt{INV}\langle f', \mathtt{Q}, \mathtt{P}\rangle \quad \mathtt{trans\ P}}{\mathtt{INV}\langle f|||_e f', \mathtt{Q}, \mathtt{P}\rangle}$$

Fig. 5. Invariant inference rules

postrelations of f and f', provided that either f preserves `R_n` or f' preserves `R_k`. If there is an access violation after f, the computation stops and `R_n` must hold. Otherwise, f' will execute and `R_k` must hold. Thus, the postrelation is the union of `R_n` and `R_k`.

Theorem 6. *All assertions* $\{f : R \rightarrow R'\}$ *derivable according to the inference rules in Figure 4 are valid.*

Non-exfiltration Similar to the non-infiltration proof, the proof of non-exfiltration uses a sound but incomplete inference system, this time concerning computation invariants of the following shape:

$$\mathtt{INV}\langle f, \mathtt{Q}, \mathtt{P}\rangle = \forall s, t.\ \mathtt{Q}\ s\ \wedge f\ s = \mathtt{ValueState}\ \ a\ t\ \implies \mathtt{P}\ s\ t\ \wedge\ \mathtt{Q}\ t\ .$$

That is, if Q holds of the prestate then `P` holds of the prestate-poststate pair, and `Q` of the poststate. We use a simple collection of inference rules to prove `Q` and `P` , shown in Figure 5. In this figure, `refl P` and `trans P` respectively state that `P` is reflexive and transitive. For non-exfiltration we need to prove that `unmodified i` is satisfied during the execution of each instruction both when it ends in user mode and when switching to privileged mode. A prerequisite for this is that the MMU is configured correctly during computation. To prove the non-exfiltration property, we check $\mathtt{INV}\langle \mathtt{next}, \mathtt{mmu\_setup\ i}, \mathtt{unmodified\ i}\ \ \rangle$.

Theorem 7. *All assertions* $\mathtt{INV}\langle f\mathtt{,Q,P}\rangle$ *derivable according to the inference rules in Figure 5 are valid.*

Privileged Constraints The final goal is to prove that `next` establishes the relation `priv_const`, a conjunction of primitive constraints `P`. Since the primitive constraints do not always hold during computations in privileged mode, the inference rules of Figure 5 are generally not able to prove this property. To make verification tractable, we prove primitive constraints locally at the point in the monadic computation where it is established and then use a set of inference rules to infer its correctness for the entire computation. We illustrate the proof

```
take_svc_exception  = IT_advance  ≫=nav
   (λ u.(read_reg 15w |||nav exc_vector_base |||nav read_cpsr |||nav
              read_scr |||nav read_sctlr )≫=nav
     (λ(pc,ExcVectorBase,cr,scr,sctlr).
        (condT (cr.M = 0b10110w) (write_scr  (scr with NS := F))  |||nav
         write_cpsr (cr with M := 0b10011w)) ≫=nav
         (λ (u1,u2). (write_spsr cr |||nav
           write_reg 14w (if cr.T then pc - 2w else pc - 4w) |||nav
           (read_cpsr ≫=nav
            (λ cr'.write_cpsr (cr' with
                           <| I := T; IT := 0b00000000w;J := F;
                              T := sctlr.TE; E := sctlr.EE |>))) |||nav
             branch_to (ExcVectorBase + 8w)) ≫=nav unit4)))
```

Fig. 6. The HOL4 code for switching to `svc` mode [4]

using an example. In the ARM model, all computations which lead to a privileged mode `m` end by a computation called `take_m_exception`. Figure 6 shows the function `take_svc_exception` for switching to supervisor mode. Let this computation start in state `s1` and end in state `sn`. Consider the primitive constraint $\mathtt{P_{psr}}$ stating that `SPSR_svc` of the final state `sn` must be equal to `CPSR` of the initial state `s1`. Let `t` and `t'`, respectively be the initial state and final state of `write_spsr cr` and `m` be the mode of `t'`. The computation `write_spsr cr` writes the value of free variable `cr` into `SPSR_m` and establishes the property $\mathtt{P'_{psr}} \overset{\text{def}}{=} \mathtt{t'.psrs(SPSR\_m)} = \mathtt{cr}$. We call `write_spsr cr` a $\mathtt{P'_{psr}}$-establisher. A computation g is `P`-establisher, if independently of its input state, `P` holds in its output state, i.e.

$$\mathtt{P{-}establ}(g) = \forall s, a, t.\ g\ s = \mathtt{ValueState}\ a\ t\ \wedge\ \mathtt{nav}\ t\ \Longrightarrow\ \mathtt{P}\ t$$

We can prove that the block starting from `write_spsr cr` establishes $\mathtt{P'_{psr}}$ as well, because the rest of the computations of this block does not modify this property. Then we can prove that the free variable `cr` takes the value `s1.psrs(CPSR)`, and `m` is bound to `svc`. Thus, `sn.psrs(SPSR_svc)` = `s1.psrs(CPSR)` holds for the computation block from `write_spsr cr`. As this block is a $\mathtt{P_{psr}}$-establisher, we conclude that the computations before `write_spsr` do not influence the established property and $\mathtt{P_{psr}}$ is satisfied by `take_svc_exception`.

Figure 7 shows the `P`-establisher inference rules. These rules along with the inference rules of Figure 5 are used to prove the privileged constraints. The rule seqTS1 states that if the monadic computation f is a `P`-establisher and `P` is an invariant of f', then the sequential composition $f \gg\!=_{\mathtt{nav}} f'$ is `P`-establisher. The rule seqTS2 describes that if the monadic computation f is a `P`-establisher, then $f' \gg\!=_{\mathtt{nav}} f$ is also `P`-establisher. Similar rules are defined for the $|||_{\mathtt{nav}}$ operator.

Theorem 8. *All assertions* `P-establ`(f) *derivable according to the inference rules in Figure 7 are valid.*

$$\text{seqTS1}\,\frac{\texttt{P-establ}(f) \quad \texttt{INV}\langle f', \texttt{P}, \top\rangle}{\texttt{P-establ}(f \gg\!=_{\texttt{nav}} f')} \qquad \text{seqTS2}\,\frac{\texttt{P-establ}(f)}{\texttt{P-establ}(f' \gg\!=_{\texttt{nav}} f)}$$

$$\text{parTS1}\,\frac{\texttt{P-establ}(f) \quad \texttt{INV}\langle f', \texttt{P}, \top\rangle}{\texttt{P-establ}(f\ |||_{\texttt{nav}}\ f')} \qquad \text{parTS2}\,\frac{\texttt{P-establ}(f)}{\texttt{P-establ}(f'\ |||_{\texttt{nav}}\ f)}$$

$$\text{absS}\,\frac{\forall y.\texttt{P-establ}(f\ y)}{\texttt{P-establ}(\lambda y.f)}$$

Fig. 7. Privileged constraints inference rules

6 Implementation and Evaluation

Implementation We use the HOL4 theorem prover to verify our properties. The central assets of our work are available from [5]. We have developed a tool, ARM-prover, to automate the verification process based on the proof systems in Fig. 4 and 5. To avoid having to explore the instruction set more than once the prover actually combines the theorems 1, 2 and 3 into one.

The proof systems do not provide rules for `case` and `let` statements. These are easily handled using standard HOL4 simplification. Other monadic expressions are refined using the inference rules in Fig. 4 and 5 in a top down fashion. The proofs for "write" primitives as well as register and memory accesses in user mode are done manually, but the tool can handle some of the "read" computations directly, allowing to prove a large share of the workload automatically.

A particular difficulty concerns binding. When a binding expression $f_1 \gg\!=_{\texttt{nav}} f_2$ is decomposed the return value of f_1 becomes unbound in f_2. To handle this we simplify computations by embedding more information before calling the prover, using some auxiliary lemmas. For example, the following formula states that `cpsr` in computation `H` following `read_cpsr` can be substituted by the CPSR in prestate `s` with mode `m`.

```
(mode s = m) ⇒ (read_cpsr ≫=nav (λcpsr. H(cpsr))) s =
         (read_cpsr ≫=nav (λcpsr. H(s.psrs(CPSR) with M:=m))) s
```

For the case that an instruction leads to a privileged mode, the last execution phase of the instruction, called switching phase, is in privileged mode. However, the privileged constraints first have to be established over the course of several steps and do not hold from the beginning. Since we can not use the ARM-prover tool to prove them automatically, we prove the privileged constraints for the switching phase manually.

Evaluation The Cambridge model of ARM is 9 kLOC. In addition to the ARM model, we rely mainly on the relatively small inference kernel of the HOL4 theorem prover, our MMU extension (about 180 lines of definitions) and the formulation of the discussed properties (about 290 lines). The entire proof script has a length of about 13 kLOC and needs roughly an hour to run on an Intel(R) Xeon(R) X3470 core. We invested about one person year of effort into this work.

7 Related Work

Several recent works address kernel verification. Some target information flow properties [7,12,15,18], based on variants of noninterference [11]. Other work establishes a refinement relation between kernel code, in some representation, and an abstract specification. For the seL4 microkernel this was first performed for its C implementation [13] and is now extended to binary level [20]. As is the case with most refinement/simulation-based approaches, this work does not address information flow. In recent work on seL4 verification, Murray et al. [14,15] present an unwinding-style characterization of intransitive noninterference. They introduce a proof calculus on nondeterministic state monads that is similar to that of this work. Their assertions are more general, however our proof rules cover several monadic operators and statements. In addition, we introduce rules to prove properties about executions that relate the final state of a computation to its initial state.

Alkassar et al. [6] describe the emulation of a simplified MIPS machine in C. The emulator allows the use of VCC to automatically check that every reachable state of a guest on a hypervisor is also reachable when the guest is running on a completely isolated machine. The C emulator has been adopted to verify parts of the hypervisor that mix C and assembly [17], and allows unknown user processes to be considered. Information flow properties are not considered, however.

Wilding et al. [21] formally proved exfiltration, infiltration and mediation theorems for the partitioning system of the AAMP7G microprocessor in ACL2. The hardware architecture differs from the one of ARM in several points, such as that there are no user-visible registers or that AAMP7G itself functions as a separation kernel. Proofs were performed using abstraction/refinement techniques and address kernel microcode. The verification led to a MILS certificate on Evaluation Assurance Level 7.

The ARMor system [22] sandboxes applications on ARM and provides formal verification of memory safety and control flow integrity, using the Cambridge HOL4 ARM model. Its software fault isolation does not use hardware features such as an MMU, but uses instead rewriting and subsequent verification of the compiled programs. This implies performance overhead, limitations on supported programs and verification processes in the extend of hours for each program. Furthermore, ARMor only establishes memory write protection; neither confidentiality nor protection of privileged registers is addressed.

Most works on kernel verification address handler code only and do not consider user mode execution. In a few cases [6,19] user mode execution is considered, but without justification in terms of concrete processor access modalities. The main contribution of our work, over and beyond the above works, is that we attempt to justify the critical assumptions on processor level information flow in user mode execution through analysis at the level of a formalized ISA model.

Heitmeyer et al. [12] introduce non-exfiltration, non-infiltration, kernel integrity and data/control separation properties to verify a separation kernel. Since we focus on user-mode execution, those properties apply only partially here. Our non-infiltration property is the same as in [12], but the non-exfiltration property in our work covers both their kernel integrity and non-exfiltration.

8 Conclusion

We introduced and proved several security properties including a non-exfiltration, a non-infiltration and a safe switching property for user mode executions on the ARM architecture, using the Cambridge HOL4 ISA model. A logical framework based on (relational) Hoare logic has been developed for the analysis, supported by a tool, ARM-prover, which helps automate the proof. The ARM-prover can be used to prove general invariants about the ARM model (i.e., statements that need to hold at each execution point). We are planning to continue the development of the ARM-prover to improve automation further and cater for more general proof tasks.

Our results concerning register contents are generally valid and with small adaptations applicable in isolation verification of other hypervisors, separation kernels, and operating systems. Statements on memory safety depend on our specific setup. A reformulation that is independent of concrete MMU configurations should require a minor effort and is planned for future work.

The HOL4 model of ARM supports a partial coprocessor model. We made the assumption that the access to coprocessors via dedicated instructions is always denied in user mode. To have a more precise analysis and cover all possible side channels, a more comprehensive model of the available coprocessors involving all registers, the coprocessors' behavior and an acceptance/rejection-mechanism for register reads and writes that follows the specification is required. During context switches kernels need to mediate coprocessor registers user-accessible by dedicated coprocessor instructions. All other coprocessor registers are guaranteed to be non-modifiable in user mode. However, kernels must not introduce information flow from non-active processes to the coprocessor registers that are part of the present ARM model, since those might influence user mode execution.

Instructions that are underspecified ("unpredictable") in the ARM Architecture Reference Manual (ARMARM) are problematic. The ARM specification states that "*unpredictable* behavior must not perform any function that cannot be performed at the current or lower level of privilege using instructions that are not *unpredictable*"[3]. In one interpretation of this statement, theorems 2, 3 and 4 are valid on unpredictable instructions as well. In general, this is not true for non-infiltration. Yet, ARMARM requires further that "*unpredictable* behavior must not represent security holes" [2]. This formulation is very vague. However, we make the assumption that non-infiltration is preserved. In fact, we argue that the security properties we have presented provide manufacturers of ARM processors with a precise description of secure behavior for unpredictable cases.

Acknowledgments. Work supported by framework grant "IT 2010" from the Swedish Foundation for Strategic Research.

References

1. ARM TrustZone technology. `http://www.arm.com/products/processors/technologies/trustzone.php`.
2. ARMv7-A architecture reference manual, issue B. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b`.
3. ARMv7-A architecture reference manual, issue C. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c`.
4. HOL4. `http://hol.sourceforge.net/`.
5. PROSPER project. `http://prosper.sics.se/`.
6. E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *VSTTE*, volume 6217 of *LNCS*, pages 40–54. Springer, 2010.
7. G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In M. Butler and W. Schulte, editors, *FM*, volume 6664 of *LNCS*, pages 231–245. Springer, 2011.
8. M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, 2013.
9. L. Duflot, D. Etiemble, and O. Grumelard. Using CPU system management mode to circumvent operating system security functions. In *Proc. CanSecWest*, 2006.
10. A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *LNCS*, 2010.
11. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
12. C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Trans. Softw. Eng.*, 34(1):82–98, 2008.
13. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an os kernel. In J. N. Matthews and T. E. Anderson, editors, *SOSP*, pages 207–220. ACM, 2009.
14. T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429. IEEE Computer Society, 2013.
15. T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In C. Hawblitzel and D. Miller, editors, *CPP*, volume 7679 of *LNCS*, pages 126–142. Springer, 2012.
16. M. O. Myreen, A. C. J. Fox, and M. J. C. Gordon. Hoare logic for ARM machine code. In F. Arbab and M. Sirjani, editors, *FSEN*, volume 4767 of *LNCS*, pages 272–286. Springer, 2007.
17. W. Paul, S. Schmaltz, and A. Shadrin. Completing the automated verification of a small hypervisor-assembler code verification. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *SEFM*, volume 7504 of *LNCS*, pages 188–202. Springer, 2012.
18. R. J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. 2010.

19. J. Rushby. Formally verified hardware encapsulation mechanism for security, integrity, and safety. Technical report, DTIC Document, 2002.
20. T. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, 2013.
21. M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. Springer US, 2010.
22. L. Zhao, G. Li, B. De Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. In *Proceedings of the International Conference on Embedded Software*, EMSOFT 2011, pages 289–298, 2011.